Adapting and optimising Fluidity for high-fidelity coastal modelling

Adapting and optimising Fluidity for high-fidelity coastal modelling

Angus C.W. Creech[a], Adrian Jackson[b,*], James R. Maddison[c]

[a]*Institute of Energy Systems, University of Edinburgh*

[b]*EPCC, University of Edinburgh*

[c]*School of Mathematics and Maxwell Institute for Mathematical Sciences, University of Edinburgh*

Abstract

Work undertaken to improve the performance of Fluidity, an open-source finite-element computational fluid dynamics solver from Imperial College London, for both general computational fluid dynamics and tidal modelling problems is outlined. Optimising the general computational structure of Fluidity, along with work to improve the data decomposition and parallel load balancing enabled simulations to be run over **three times faster** than with the original code, even when using thousands of computational cores. This changes the level of detail at which fluids problems can be studied with Fluidity, and impacts upon research that examines high Reynolds number turbulent flows. This is of particular relevance in areas such as engineering aerodynamics, wind energy, marine energy, and environmental or pollution modelling.

Keywords: Fluidity, Load Balancing, Mesh Decomposition, Code Optimisation, Parallel Performance

1 Introduction

Energetic tidally-driven flow in coastal areas such as straits represents a challenge for high fidelity simulation, as the turbulent flow processes to be modelled range from less than 1m to greater than 1km in scale (eg. tidal jets). Furthermore, such turbulence is highly anisotropic, in a domain that may be only 100-200 metres deep, but have a horizontal surface area (the sea surface) of 1000s of km^2. Coastal domains also have an irregular shape,

* Corresponding Author
Email address: a.jackson@epcc.ed.ac.uk

with undulating bathymetry and complex coastlines, making them well suited to unstructured mesh approaches used in finite-element computational fluid dynamics (CFD) software such as Fluidity.

Tidal flow modelling has the potential to support efficient energy production, through optimisation of tidal turbine design and placement [1][2][3][4], and to inform on the environmental impact of aquaculture [5][6] and hazards such as algal blooms [7] through modelling of tidal flows.

High fidelity simulation is essential to ensure that local scale issues can be sufficiently resolved to enable local flow features, or local placement issues, to be properly modelled and investigated, eg. resolution must be sufficient to enable individual turbine interactions to be modelled, when turbines are likely to be tens of meters apart, or to enable fish farm installations, which are of the order of $40m^2$, to be included in flow simulations and inform placement and run-off discussions for existing and new farms.

Therefore, there is the requirement for high fidelity and high resolution simulations, including energetic flow features, on large scale simulations. Excluding Direct Numerical Simulation (DNS), simulations at high Reynolds number require the use of an appropriate turbulence model. One option is Large Eddy Simulation (LES) [8][9][10], which can capture high levels of detail in the transient turbulent flows of environmental fluid mechanics problems [11]. It is being increasingly applied to large problems, such as wind farm modelling [12][13][14] and coastal modelling [15][16]. Many of these problems have highly anisotropic domains, often with anisotropic grids – therefore an LES algorithm using tensors for subgrid eddy viscosity and length scale metrics are desirable, as a scalar SGS viscosity applied to high aspect ratio elements results in excessive turbulent diffusion [17][18].

In this paper we outline work undertaken to adapt and optimise a code [19] that implements such an approach, Fluidity, to significantly improve performance, and thereby scientific output, on modern computing hardware. The resulting modified variant of the code is available from [20]. In the remained of the paper we will outline the code to be optimised, the computational hardware optimising for, and the optimisation techniques applied. The paper concludes with a summary of findings.

2 Fluidity

Fluidity [21][22] is an open source finite-element computational fluid dynamics (CFD) code that solves the incompressible non-hydrostatic Navier-Stokes momentum equation and thermodynamic continuity equation. It is capable of modelling free surfaces and is designed for irregularly shaped, unstructured grids. Fluidity can use

both OpenMP and MPI for parallel computation, and has been shown to scale to large numbers of cores on high performance computing (HPC) systems [23].

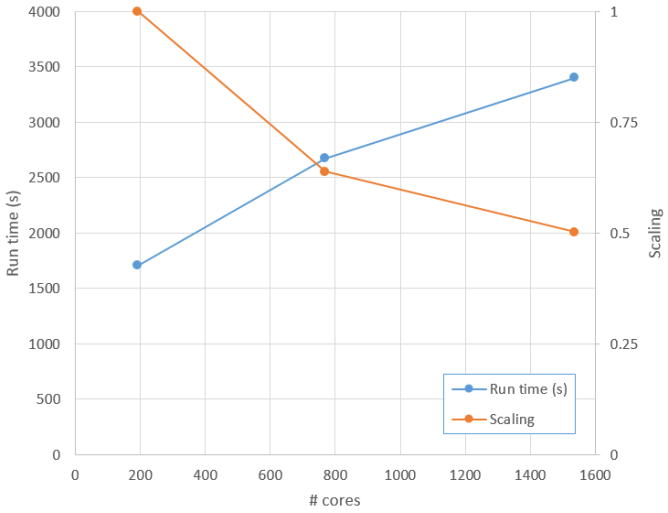


Figure 1: Weak scaling of Fluidity for representative tidal modelling simulations. Ideal performance would be a scaling of 1 and a flat runtime as the number of cores used increases.

Finite-element CFD modelling requires the discrete representation of the velocity and pressure fields using a finite set of basis functions defined by the computational mesh. Common approaches are the Continuous Galerkin (CG) method, where the basis functions describe a continuous field, and the Discontinuous Galerkin (DG) method, where the fields are only continuous within an element.

Within Fluidity there can be a separate basis for velocity and pressure, defining finite *element pairs*. Pairs frequently used in Fluidity are P1-P1, which is a first order (locally linear) CG velocity, first order CG pressure, and P1DG-P2[24][25][26], which is first order DG velocity, and second order (locally quadratic) CG pressure. These element pairs each have differing properties regarding stability and suitability for certain flows. To locally increase the accuracy of a finite-element solution, mesh resolution can either be increased by *h*-adaptivity techniques, or via *r*-adaptivity moving degrees of freedom to areas of interest, or by a combination of the two [27][28].

Another option is to vary the order of the elements in a mesh, in what is known as *p*-adaptivity, giving a mixed mesh which could contain 1st, 2nd, 3rd … (etc.) order elements within the same mesh. Unfortunately, *p*-adaptivity can have a substantial computational cost if not implemented carefully. As every element in a mesh has an arbitrary type and order, the basis functions are also arbitrary, as are the sizes of the matrices that represent

them. Fluidity has a partial implementation of *p*-adaptivity, which has led to several design-related performance issues, as acknowledged in a previous optimisation work [23].

Firstly, compilers are prevented from perform compile-time optimisation such as loop unrolling/vectorisation, as inner loops have run-time defined length. Secondly, as assembly loops are designed to work with matrices whose sizes vary, large numbers of allocations and deallocations are required per iteration. This restricts the ability of compilers to inline, or further optimise, these functions.

Fluidity requires large amounts of computational time, and does not scale well for representative simulations (see Figure 1), making it difficult to perform substantial investigations of turbulent flow for tidal turbines to enable estimation of tidal power potential, optimisation of turbine configurations, or analysis of the impact of turbine placement on sea beds.

3 Computing Hardware

Modern computing hardware is reliant on fast, processor local, memory, namely caches, to achieve good compute performance. This reliance arises from the disparity in performance between processors and main memory, with processors capable of executing 100s to 1000s of instructions in the time it takes to load a unit of data from main memory.

Caches exploit the principles of temporal and spatial data locality that most programs exhibit. That is to say, most programs will either use the same piece of data multiple times for computations within a given time frame (temporal locality), or they will use data located close (in memory) to data that has been recently used (spatial locality).

Storing recently used data in a cache, and loading a larger region of data into cache when a specific piece of data is required to complete an instruction rather than simply loading that piece of data, enables processors to achieve higher performance for a wide range of programs whilst still operating with a large and slow main memory system.

This being the case, it is essential for good computational performance that the implementation of an algorithm is structured in such a way as to make good use of caches. There are a number of algorithmic techniques that

can be used to increase efficient use of caches by programs; examples such as cache blocking, strip mining, and pre-fetching are widely exploited by applications.

Furthermore, modern processors are heavily reliant on vector functionality, hardware that can perform the same mathematical operation on multiple pieces of data at the same time, to achieve peak performance, particular for floating point operations. As computational simulation applications, such as CFD, are generally dominated by floating point calculations, ensuring that such applications can effectively utilise vector hardware is key in ensuring that those applications achieve good performance.

Vectorisation is generally achieved through compiler optimisation techniques: compilers identifying sections of codes that can utilise vector hardware and generating the appropriate vector instructions for those code sections. These sections of codes need to repeat the same operation over a range of data, and therefore code loops (generally the core computational kernels for scientific simulation codes) are the main target for vectorisation by compilers. However, compilers have to be conservative when identifying code that can be vectorised, as incorrectly vectorising code can produce incorrect results.

Therefore, compilers struggle to vectorise a wide range of code, meaning best performance is not achieved on modern computing hardware for those codes. Issues that can cause a compiler to not identify vectorisable code include loops where the loop bounds are not evident at compile time, data structures not being aligned to memory boundaries, small loops, pointers in loops and dependencies between loop iterations.

4 Initial Performance

We outline the performance of the code prior to optimisation to enable a proper analysis of performance improvements and motivate the reasons for the optimisations implemented. Figure 1 demonstrates the performance of Fluidity with a weak scaling test case across a number of MPI process counts. This test case is based upon an idealised, rectilinear channel with a logarithmic velocity profile as a Dirichlet condition at the inlet, an open boundary at the outlet, a free-slip boundary condition on the top and sides, and a quadratic drag on the bottom. No normal flow was allowed on the top, bottom and sides.

The channel measured 1000m x 250m x 50m, as shown in Figure 2. The linear systems associated with the momentum equations are solved used the GMRES solver with SSOR preconditioning; the pressure associated with a project step is solved with Conjugate Gradient using SSOR for preconditioning via PETSc [31]. The

simulation was set to run for 100 time-steps, with 2 Picard [29] iterations per timestep. This meant the DG assembly would be called 200 times, which was deemed sufficient for performance measurements.

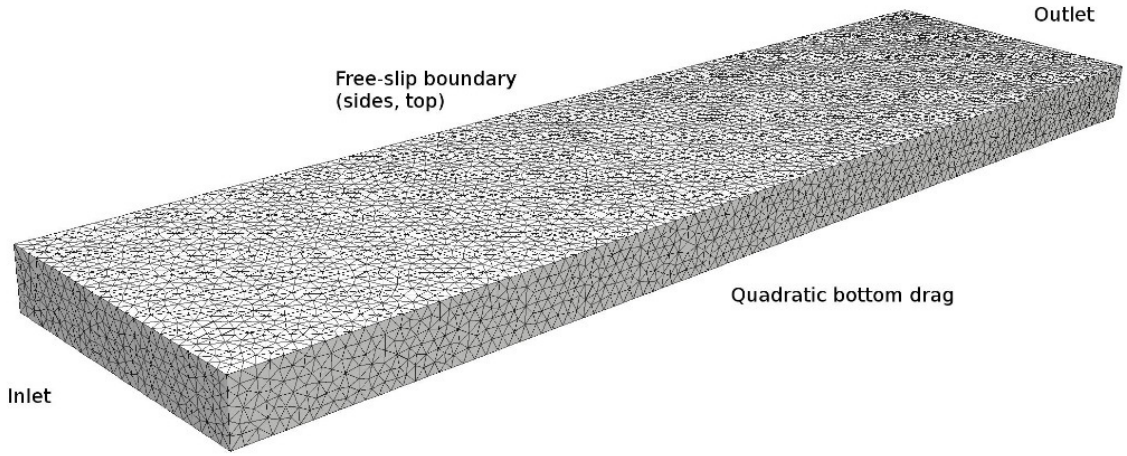


Figure 2. Channel test case for DG optimisation, here showing the mesh for 24 MPI processes. The mesh was generated using the Gmsh [30] meshing utility.

MPI tasks	**Cores (2xOpenMP)**	**Nodes**	**Max. Δx (m)**	**Elements**	**Elements / MPI task**
96	192	8	4.05665	968 664	10 090
384	768	32	2.4730	4 169 234	10 857
768	1536	64	1.9309	8 569 453	11 158

Table 1. Soft-scaling configurations for profiling on ARCHER.

Testing and benchmarking in this paper was performed on the UK National HPC Service, ARCHER [32]. This is a 118,080 core Cray XC30 system, with 24 cores (two Intel Xeon 2.7 GHz, 12-core E5-2697v2 processors) and 64 GB per node, and the Cray Aries network. Fluidity and associated libraries were compiled with the GNU 4.9.3 Fortran compiler, using the `-O2 -ffast-math -funroll-loops -march=native` optimisation flags.

For profiling on ARCHER, a weak-scaling approach was used, since partitioning the problem into ever-smaller chunks results in tiny partitions and a problem dominated by communication overhead. Weak scaling was calculated from the following equation, ie. $s = t_1/t_N$, where t_1 is the time taken to completion on 1 processing unit, and t_N the time taken on N processing units.

However, due to memory and processor limitations, it is difficult to run a meaningful problem small enough for 1 core in this context, we use t_{192} instead as our base case. The complexity of the mesh was controlled through the maximum allowable size of the elements (max. Δx), and whilst the number of MPI tasks scaled exponentially (see Table 1), the number of elements was kept approximately constant.

The performance of Fluidity for these test runs was profiled with Allinea MAP, with the most expensive high level routines shown in Table 2. Note, that any direct PETSc library [31] calls or wrappers have been removed, and that these figures are inclusive (i.e. routines times include the time of any routine called from with that routine).

Fluidity, like many applications, requires some initial setup time for simulations, incurred once at the beginning of a simulation. We present the profiling data with and without that setup cost. It is evident that this setup phase can change the performance figures by more than 1%.

To ensure this setup cost does not influence our timing data, as we are running short simulations for this performance analysis where the setup costs will disproportionally impact performance compared to normal (much longer) production simulations, only the percentage runtimes without setup shall be used in the rest of the paper. The data in Table 2 is also represented in Figure 3.

	% runtime (with / without setup)					
# cores	**192**		**768**		**1536**	
Function name						
solve_momentum	90.6	92.20	91.6	92.3	93.2	94.1
correct_pressure	37.6	38.4	47.0	48.3	52.7	55.9
construct_momentum_dg	28.3	29.3	20.3	21.1	17.1	17.6
construct_momentum_element_dg	27.8	28.7	19.9	20.8	16.8	17.3
advance_velocity	16.2	15.8	17.1	15.5	17.1	14.9
construct_momentum_interface_dg	13.5	14.0	9.5	9.9	8.1	8.3
subcycle_momentum_dg	13.4	13.0	13.7	11.8	14.5	12.3
assemble_cmc_dg	6.3	6.6	5.0	5.2	4.2	4.2
local_assembly_cdg_face	6.6	6.9	4.6	4.8	3.9	4.0
calculate_courant_number	4.9	5.4	4.1	4.4	4.2	4.3

Table 2. % run times of unoptimised code

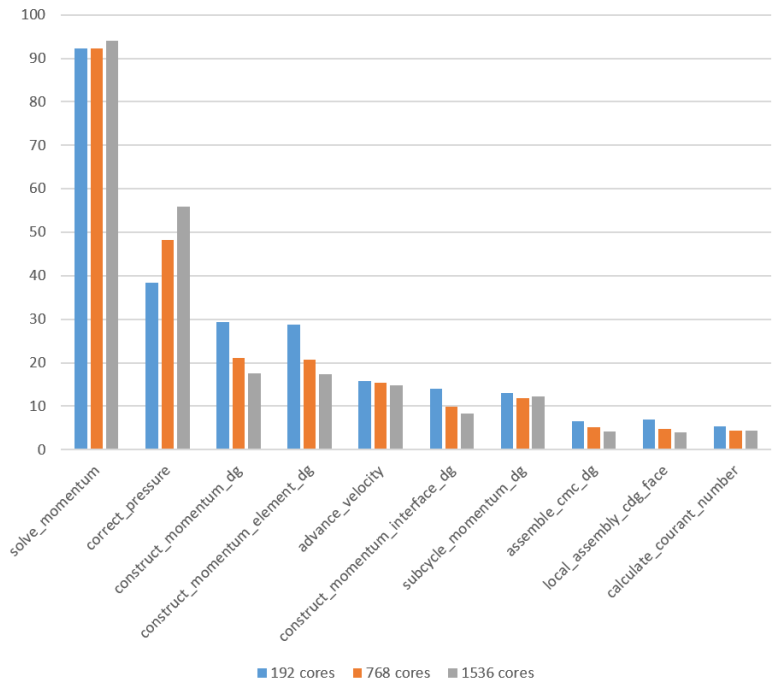


Figure 3. Percentage runtimes for unoptimised code

The main subroutine, *solve_momentum* dominates the calculations, representing over 90% of execution time: it calls all of the proceeding functions. *correct_pressure* is nearly 38-55% of execution time, virtually all of which is spent inside PETSC function calls. The next largest valid target was *construct_momentum_dg*. This is what assembles the global momentum matrix for solution and associated right hand-side (RHS). It does this calling by *construct_momentum_element_dg* (CME_DG for short) within a tight loop, where it spends about 98% of its time. Aside from taking a substantial amount of execution time (~17–30%), two functions called from CME_DG, *construct_momentum_interface_dg* and *local_assembly_cdg_face*, occupy two thirds of that time.

5 Optimisations

Given the costliness of the simulations we are interested in enabling, and the performance characteristics of modern hardware, there are a number of optimisations that can be undertaken within the code to ensure that our target simulations can perform as close to optimally as possible.

In this section we outline these optimisations and discuss the performance benefits they have facilitated. These optimisations have been implemented whilst maintaining the original code functionality.

5.1 Subroutine optimisation

Fluidity is designed to allow one to three spatial dimensions (plus time), four different DG viscosity schemes (Bassi-Rebay, Interior Penalty, Arbitrary Upwind, and Compact Discontinuous Galerkin), along with many other options that affect the finite element calculations undertaken. Currently the code supports this by checking the features currently being used when it is assembling the global momentum matrix and RHS. However, these features do not change throughout a given simulation, so it is possible to check these features once, for a given simulation, and then simply use the required functionality during the simulation.

To enable this, we created tailored versions of the core computational routines that our simulations of interest use (based on profiling of the code). These tailored versions had the different choices that could be made for each element hard coded, meaning the compiler can efficiently vectorise and optimise these kernels. This functionality is combined with compile time checking of the input file for a simulation to choose the correct optimised routines for a given simulation, enabling an optimised executable to be generated. The executable also contains the generic versions of the routines that will be called at runtime for any parts of the simulation using functionality that has not yet be optimised, enabling both optimised and non-optimised routines to be exploited in a given simulation.

In more details, looking at *construct_momentum_dg,* which assembles a global momentum matrix and associated RHS for solving, it loops over each element in the local submesh, creating a local matrix which it adds to the global matrix and RHS as it progresses. This functionality is already parallelised using OpenMP, distributed work over threads. The basic outline of execution in *construct_momentum_dg* is:

```
construct_momentum_element_dg:
Set relevant options from FLML tree
For each element e in submesh:
      call construct_momentum_element_dg:
            For each face f in element e:
                  call construct_momentum_interface_dg:
                        call local_assembly_cdg_face
```

The original design of *construct_momentum_dg* had each of these options being parsed at run-time within *CME_DG*, for each element. Moreover, there are tens of smaller arrays that are defined and allocated/deallocated at run-time, for each element, and for each face of each element.

Lastly, there were many finite-element utility functions that consist of little more than matrix multiplication, but each of which allocates and deallocates small temporary work arrays upon being called and when being exited.

What is important to note here is that:

1) For each core, there are many hundreds of thousands of small memory allocations and deallocations per iteration for a reasonable-sized problem. This could easily results in millions of memory allocations for each multi-core processor in a modern compute node
2) The sizes of all of these small arrays do not change over the course of the simulation, as Fluidity only supports one type of local element basis function per discrete function space.
3) None of the element discretisation options change over the mesh for a given simulation.

From this, it was clear that compile-time parsing of these options could provide an answer. This would allow:

1) Compiling out of unnecessary option parsing using #ifdef pre-processor directives
2) Compile-time defining of common size parameters through #define directive, eg. number of dimensions and number of element nodes, which would allow static allocation of all of the small arrays
3) Compile-time vectorisation of small, tight loops (eg. looping over dimensions, nodes) by using #define.

The first optimisation task was the construction of *construct_momentum_elements_dg_opt*, a per-element local matrix assembly routine which relies on compile-time definitions for dimension, element quadrature, number of element nodes, etc. Through this, many further optimisations were undertaken to improve the performance over the original unoptimised subroutine, *construct_momentum_element_dg*. These were, namely:

- Conversion of all small dynamic array allocations to static allocations
- Optimisation of tight loops to use compile-time length definitions, to allow compile-time vectorization of the loops
- Inlining of calls to finite-element utility subroutines
- Inlining of code element face assembly subroutines, such as *construct_momentum_interface_dg*. All code was moved inside *construct_momentum_elements_dg_opt* and all dependent arrays declared statically there.

- Rearrangement of decision logic to minimise expensive recalculation of array values.
- Addition of logic in *construct_momentum_elements_dg* to only call optimised code if run-time element configuration for dimension, quadrature, etc. matches compile-time configuration; otherwise, it runs non-optimised original *construct_momentum_element_dg* code.

The second task was the creation of command-line tools to parse the simulation options file (user options file [33]) for discretisation options, and generate a header file which contained compile-time definitions of element dimension, number of faces, number of element nodes, etc. This can then be used to compile the optimised Fluidity routines.

5.2 Optimised results

We tested these optimisations on the test channel test case, with results presented in Figure 4. This shows the overall run times and speed up for the whole simulation, using the assembly-optimised code. What is clear is that, even though only the DG assembly subroutines have optimised, there has been a substantial overall improvement in performance. From 192 cores (8 compute nodes), we see a 36% speed up; when this is scaled up to 1536 cores (64 nodes), we still see a 19% speed up. This perhaps can be expected, as DG assembly does not rely on any MPI communication, which can be expected to dominate the rest of the program at very high core counts. It is also clear that the scaling of unoptimised and optimised versions follow a similar progression.

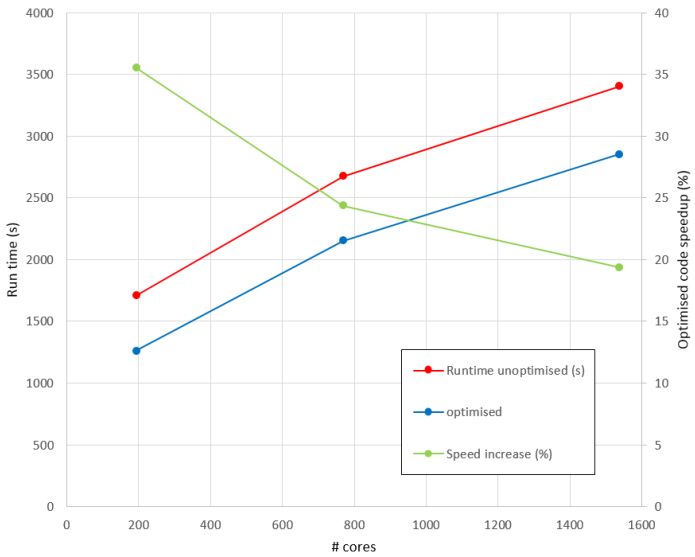


Figure 4 Optimised versus unoptimised run times, and speed up. Note that, even at the highest core counts, just optimising the DG assembly code has resulted in a 19-24% performance increase overall. This would be higher in longer simulation runs

More detail can be seen in Table 3 which shows the percentage run times of the most heavily used subroutines in the optimised code. As before, all PETSc calls and direct wrappers have been removed. The subroutine for *construct_momentum_interface_dg* was manually inlined, and so it no longer appears in the profiler log; *local_assembly_cdg_face* is now so quick it barely registers in the profiler timings. As expected, *construct_momentum_dg* now occupies substantially less of the execution time: for 192 cores, it is **2.69x** less of overall execution time; at 1536 cores, it is **3.03x** less.

	% runtime (without spin-up)		
# cores	**192**	**768**	**1536**
Function name			
solve_momentum	89.8	92.7	99.3
correct_pressure	54.0	64.6	69.7
advance_velocity	13.7	11.6	10.6
subcycle_momentum_dg	11.3	9.9	8.9
construct_momentum_dg	10.9	8.0	5.8
assemble_cmc_dg	8.9	6.8	5.2
construct_momentum_element_dg	8.4	6.4	5.0
construct_momentum_interface_dg	-	-	-
local_assembly_cdg_face	-	-	-
calculate_courant_number	3.7	3.0	2.5

Table 3. % run times of optimised code. The blank indicate code that no longer exists or registers in the profiler output.

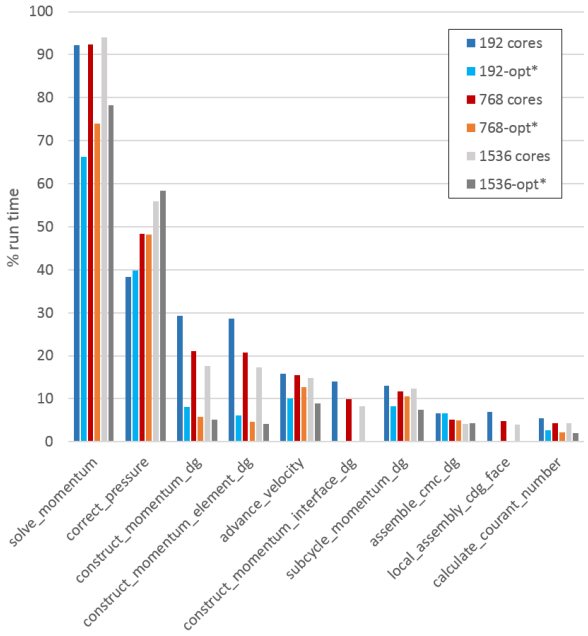


Figure 5. Comparison of % run times for unoptimised code, versus optimised code scaled by relative run time for each case. The asterisk (*) indicates a scaled run time

As the profiling tool used does not conveniently give the absolute run times of individual routines, to give better idea of the impact on run times of the DG assembly optimisation on each of the major subroutines, Figure 3 was repeated but with the optimised percentage run times scaled by the relative change in the runtimes, ie. with a scaling factor $s = R_{opt}(N)/R_{unopt}(N)$ where $R_{xxx}(N)$ is the overall run time for N cores for case *xxx* (ie. *opt* or *unopt*). This would give a scaled percentage run time of:

$$R^*(N) = s(N)R(N)$$

The scaled percentage run times of the optimised code can be thought of as the run time relative to that of the unoptimised code.

The graph of the scaled results are shown in Figure 5. As expected, DG assembly has a much shorter absolute run time: at 96 cores, there is a **3.64x** absolute speed up; at 1536 cores, the absolute speed up is **3.39x**. Some subroutines independent of the assembly code (eg. *correct_pressure* and *assembly_cmc_dg*) have almost the same execution time before and after the DG optimisation, but others (*advance_velocity*) show a substantial improvement.

5.3 Load balancing and decomposition

We observed that the meshes generated and used by Fluidity are not well ordered. For this type of simulation Fluidity extrudes the 2D mesh (representing open boundaries and coastlines) downwards into a 3D mesh matching the bathymetry of the simulation being undertaken. By re-ordering the numbering of mesh elements on the 2D mesh (see Mesh Reordering section), we can ensure that the extruded 3D mesh has elements where neighbours have similar element numbers, meaning processes are likely to own the neighbouring elements they need data from when performing calculations on a given element. This, combined with adding load balancing to the mesh partitioning functionality in Fluidity, taking into account the extruded 3D mesh size when partitioning the 2D mesh across processes, provides significant improvements in the parallel performance of Fluidity for tidal simulations.

This improvement allows the code to more effectively balance mesh sizes across subdomains, even when there are spatially variable numbers of element layers. Subdomain mesh sizes from the tidal test case before and after this improvement are shown in Table 4.

Case	Min.	Max.	Imbalance	Mean	Standard dev.
Pre-optimisation	10088	64538	6.398	16058.979	7727.234
Load balancing	10930	25365	2.321	16369.620	2803.807

Table 4. Statistics on elements for partitioned subdomains before and after load balancing enhancements.

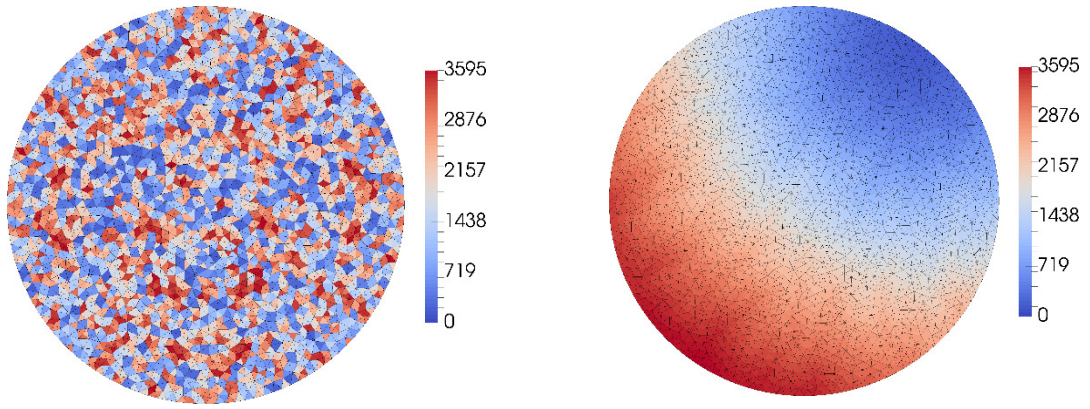
It can be seen that before optimisation, the largest subdomain contained almost 6.4x the elements of the smallest subdomain. This severe load imbalance resulted in smaller partitions waiting on blocking MPI communications (e.g. mpi_allreduce) until the larger partitions finish. The code with the enhanced load balancing brought the imbalance down to 2.3x – approaching one third of the pre-optimised imbalance.

6 Mesh Reordering

Finite element calculations require the calculation of integrals over the computational mesh, leading to the construction of sparse matrices or vectors. Such finite element assembly is typically performed by computing integrals over each element of the mesh in turn. These latter local integrals use data for the element considered, and typically also use data for neighbouring elements. However mesh generators may create meshes which are

sub-optimal for such calculations. Further issues may be encountered in solving linear systems associated with finite element discretisations, if the sparsity patterns of the associated matrices are sub-optimal.

A pre-processing tool was written to reorder mesh data. The Gibbs-Poole–Stockmeyer algorithm[34][33][35] is used to reorder the mesh elements, based upon a reordering of the nodes in the element-element connectivity graph (with graph edges defined between mesh elements which share a common facet). The mesh vertices are similarly reordered based upon a reordering of the vertex-vertex connectivity graph. Note that a more advanced algorithm could consider a relation between the vertex and element ordering (see e.g. [36]), although here the two sets are ordered independently. The tool makes use of the SCOTCH library [37], using interfacing code derived from DOLFIN 1.5.0 [38].



a) b)

Figure 6. a) A mesh generated using Gmsh 2.10.1. The elements are coloured according to their element number indexed from zero. b) The same mesh, with elements reordered using the Gibbs-Poole–Stockmeyer algorithm[34][35] using SCOTCH 6.0.4,and using interfacing code derived from DOLFIN 1.5.0.

7 Performance Evaluation

To evaluate the performance of all the optimisations undertaken a more realistic test case is considered, chosen to be as close as possible as to the type of simulation the code has been optimized for: ie. a tidal coastal model. This was developed from a pre-existing prototype of the Sound of Islay, spanning 21.4 km east-west and 36.6km north-south as shown in Figure 7.

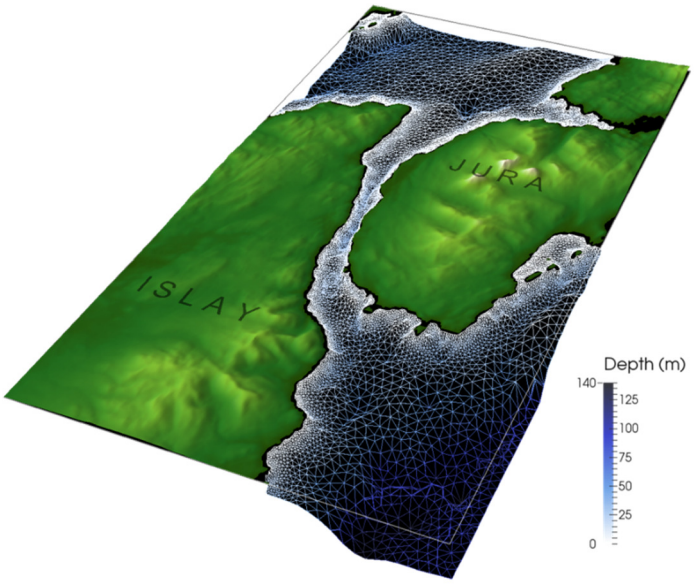


Figure 7. Sound of Islay overview

The mesh in this simulation, as with all tidal simulations within Fluidity, is a semi-structured 3D mesh. This means that an initial 2D mesh which matches the coastlines and open boundaries is extruded downwards to a depth which matches the specified bathymetry. The mesh is still composed of tetrahedra, but the tetrahedra are arranged in vertical columns.

The simulation was set to run for 100 timesteps; there were two Picard iterations for each timestep, which means the DG assembly code is called 200 times. To track the improvements that each section of optimisation work made to performance, the model was benchmarked at four stages of development: i) the code with no optimisations, ii) with DG assembly optimisation, iii) using the 2D (pre-extruded) mesh reordering tool, and iv) load balancing tweaks for extruded meshes.

The mesh configuration used for all tests is shown in Table 5.

MPI tasks	**Cores (2xOpenMP)**	**Nodes**	**Min-Max. Δx (m)**	**Elements**	**Elements / MPI task**
384	768	32	8-80	3 784 390	9855

Table 5. Realistic test case configuration for profiling on ARCHER.

Table 6 and Figure 8 highlight the effect of each successive optimisation on the performance of Fluidity on the tidal case. Interestingly, the DG assembly optimisation has an even more dramatic effect on code performance than with the unstructured 3D case – **1.24x** versus **1.74x** for the tidal testcase. Running the 2D mesh reordering

increases the performance over the pre-optimised code to **1.95x**, a further 11% increase, which is impressive considering only the 2D mesh is being reordered, and no internal Fluidity code was rewritten.

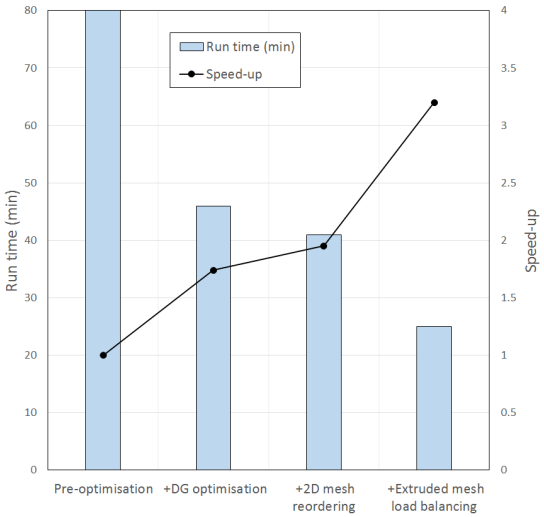


Figure 8. Graph of speed-ups for tidal case with 768 cores

Finally, as expected, addressing the load balancing issues gives an additional 64% speedup, resulting in the tidal test case running **3.2x** faster with the optimised version of Fluidity, than it did with the original, pre-optimised code. This is significant, as this changes the level of physical modelling that can be achieved with Fluidity.

Case (n768)	**Run time (min)**	**Speed-up**
Pre-optimisation	80	-
+DG assembly optimisation	46	1.74 x
+2D mesh reordering	41	1.95 x
+Load balancing optimisation	25	3.20 x

Table 6. Values for speed-ups from successive optimisations for tidal modelling case with 768 cores.

8 Summary

We have outlined the steps taken to adapt and improve a production CFD code, Fluidity, for a specific application, tidal modelling. Without changing functionality and results, but restructuring to match the computational hardware and parallel configuration being used for production simulation, we are able to reduce runtime to less than one third of the run time of the original code.

Such improvements can enable new science by allowing much larger or higher resolution simulations to be undertaken, and to enable significantly more simulations to be undertaken for the same amount of computational resources. The source code developed in this work is open source, and available at [20].

Acknowledgements

This work was funded under the embedded CSE programme of the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk). JRM acknowledges support from the Natural Environment Research Council. This work was supported by the Natural Environment Research Council (NE/L005166/1, NE/R000999/1). AJ acknowledges funding from Intel's Parallel Computing Centre programme, through EPCC's IPCC.

We would like to thank the Applied Modelling and Computation Group (AMCG) at Imperial College, the original authors of Fluidity.

9 References

[1] S. Draper, G. Houlsby, M. Oldfield, and A. Borthwick. Modelling tidal energy extraction in a depth-averaged coastal domain. *IET Renewable Power Generation*, 4:545–554, 2009.

[2] T. Adcock, S. Draper, G. Houlsby, A. Borthwick, and S. Serhadioğlu. The available power from tidal stream turbines in the Pentland Firth. *Proceedings of Royal Society A*, 2013.

[3] Z. Yang, T. Wang, T, A. and Copping, Modeling tidal stream energy extraction and its effects on transport processes in a tidal channel using a three-dimensional coastal ocean model. *Renewable Energy*, 50, 2013.

[4] S.W. Funke, P.E. Farrell, and M.D. Piggott. Tidal turbine array optimisation using the adjoint approach. *Renewable Energy*, 63, 2014.

[5] Y. Wu, J. Chaffey, B. Law, D.A. Greenberg, A. Drozdowski, F. Page, and S. Haigh. A three-dimensional model for aquaculture: a case study in the Bay of Fundy. *Aquaculture Environment Interactions*, 5:235-248, 2014.

[6] D.R. Plew. Depth-Averaged Drag Coefficient for Modeling Flow through Suspended Canopies. *Journal of Hydraulic Engineering*, 137:2, 2011.

[7] D. Aleynik, K. Davidson, A.C. Dale, and M. Porter. A high resolution hydrodynamic model system suitable for novel harmful algal bloom modelling in areas of complex coastline and topography. *Harmful Algae*, 53(3):102–117, 2016

[8] J. Smagorinsky. General Circulation Experiments with the Primitive Equations. *Monthly Weather Review*, 91:99, 1963.

[9] J.W. Deardroff. A numerical study of three-dimensional turbulent channel flow at large Reynolds numbers. *J. Fluid Mech.*, 41(2):453–480, 1970.

[10] M. Germano, U. Piomelli, P. Moin, and W.H. Cabot. A dynamic subgrid-scale eddy viscosity model. *Physics of Fluids A*, 3, 1991.

[11] W. Rodi, G. Constantinescu, and T. Stoesser. Large-Eddy Simulation in Hydraulics. *IAHR Monographs*, 2013.

[12] M. Churchfield, S. Le, P. Moriarty, L. Martınez, S. Leonardi, G. Vijayakumar, amd J. Brasseur. A Large-Eddy simulation of wind-plant aerodynamics. *50th AIAA Aerospace Sciences Meeting*, 2012.

[13] D. Yang, C. Meneveau, and L. Shen. Large-eddy simulation of offshore wind farm. *Physics of Fluids*, 26, 2014.

[14] A. Creech, W-G. Früh, and A.E. Maguire. Simulations of an Offshore Wind Farm Using Large-Eddy Simulation and a Torque-Controlled Actuator Disc Model. *Surveys in Geophysics*, 36:3, 2015

[15] F. Roman, G. Stipcich, V. Armenio, R. Inghilesi, and S. Corsini. Large eddy simulation of mixing in coastal areas. *International Journal of Heat and Fluid Flow* 31, 2010.

[16] A. Creech. A Large Eddy Simulation of an Energetic Tidal Site: The Sound of Islay. *CAIMS Annual Meeting*, 2017.

[17] A. Petronio, F. Roman, C. Nasello, and V. Armenio. Large eddy simulation model for wind-driven sea circulation in coastal areas. *Nonlin. Processes Geophys.*, 20, 2013.

[18] J.R. Bull, M.D. Piggott, and C.C. Pain. A finite element LES methodology for anisotropic inhomogeneous meshes. *Turbulence, Heat and Mass Transfer*, 7, 2012.

[19] Technical report: http://www.archer.ac.uk/community/eCSE/eCSE05-07/eCSE05-07.php

[20] CoastED: High-Fidelity Coastal Modelling. URL: https://www.github.com/coasted. *GitHub open-source repository*, 2017.

[21] C.C. Pain, M.D. Piggott, A. Goddard, F. Fang, G.J. Gorman, D. Marshall, M. Eaton, P. Power, and C. de Oliveira. Three-dimensional unstructured mesh ocean modelling. *Ocean Modelling*, 10(1-2): pp. 5-33, 2005.

[22] M.D. Piggott, G.J. Gorman, C.C. Pain, P.A. Allison, A.S. Candy, B.T. Martin, and M.R. Wells. A new computational framework for multi-scale ocean modelling based on adapting unstructured meshes. *International Journal for Numerical Methods in Fluids*, 56, pp. 1003-1015, 2008.

[23] X. Guo, G.J. Gorman, M. Lange, A. Sunderland, and M. Ashworth. Developing Hybrid OpenMP/MPI Parallelism for Fluidity-ICOM - Next Generation Geophysical Fluid Modelling Technology. *Technical report*, dCSE, 2012.

[24] C.J. Cotter, D.A. Ham, C.C. Pain, and S. Reich. LBB stability of a mixed Galerkin finite element pair for fluid flow simulations. *Journal of Computational Physics*, 228(2): pp. 336-348, 2009.

[25] C.J. Cotter and D.A. Ham, and C.C. Pain. A mixed discontinuous / continuous finite element pair for shallow-water ocean modelling. *Ocean Modelling*, 26: pp. 86-90, 2009.

[26] C.J. Cotter and D.A. Ham. Numerical wave propagation for the triangular P1DG-P2 finite element pair. *Journal of Computational Physics*, 230, pp. 2806-2820, 2011.

[27] C.C. Pain, A.P. Umpleby, C. de Oliveira, and A.J.H. Goddard. Tetrahedral mesh optimisation and adaptivity for steady-state and transient finite element calculations. *Computer Methods in Applied Mechanics and Engineering*, 190, pp. 3771-3796, 2001.

[28] M.D. Piggott, C.C. Pain, G.J. Gorman, P.W. Power, and A.J.H. Goddard. h, r, and hr adaptivity with applications in numerical ocean modelling. *Ocean Modelling*, 10, pp. 95-113, 2005

[29] R. Ford, C.C. Pain, M.D. Piggott, A.J.H. Goddard, C. de Oliveira, and A.P. Umpleby. A nonhydrostatic finite-element model for three-dimensional stratified oceanic flows. Part I: Model formulation. *Monthly Weather Review*, 132, pp. 2816-2831, 2004.

[30] C. Geuzaine and J.-F. Remacle. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11), pp. 1309-1331, 2009.

[31] S. Balay, S. Abhyankarm M.F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W.D. Gropp, D. Kaushik, M.G. Knepley, D.A. May, L. Curfman McInnes, K. Rup, B.F. Smith, S. Zampini, and Hong Zhang. PETSc, http://www.mcs.anl.gov/petsc, 2017.

[32] UK National HPC Service website. http://www.archer.ac.uk

[33] D. A. Ham, P. E. Farrell, G. J. Gorman, J. R. Maddison, C. R. Wilson, S. C. Kramer, J. Shipton, G S. Collins, C. J. Cotter, and M. D. Piggott, Spud 1.0: Generalising and automating the user interfaces of scientific computer models, *Geoscientific Model Development*, 2, pp. 33-42, 2009

[34] N.E. Gibbs, W.G. Poole Jr., and P.K. Stockmeyer/ An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix. *SIAM J. Numer. Anal.*, 13(2), pp. 236–250, 1976

[35] N.E. Gibbs, W.G. Poole Jr., and P.K. Stockmeyer. A Comparison of Several Bandwidth and Profile Reduction Algorithms. *ACM Trans. Math. Softw.* 2, 322-330, 1976.

[36] M. Lange, L. Mitchell, M.G. Knepley, and G.J. Gorman. Efficient Mesh Management in Firedrake Using PETSc DMPlex. *SIAM J. Sci. Comput.* 38(5), S143–S155. (13 pages), 2016.

[37] F. Pellegrini and J. Roman. SCOTCH: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. *Proceedings of HPCN'96*, Brussels, Belgium. LNCS 1067, pages 493-498, 1996.

[38] A. Logg and G.N. Wells. DOLFIN: Automated Finite Element Computing, *ACM Transactions on Mathematical Software*, vol. 37, 2010.